OULUN AMMATTIKORKEAKOULU

Ilkka Koiste

MEAN-PINOLLA TEHDYN JÄRJESTELMÄN YKSIKKÖTESTAAMINEN

MEAN-PINOLLA TEHDYN JÄRJESTELMÄN YKSIKKÖTESTAAMINEN

Ilkka Koiste
Opinnäytetyö
Kevät 2018
Tietotekniikan koulutusohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, ohjelmistokehitys

Tekijä: Ilkka Koiste
Opinnäytetyön nimi: MEAN-pinolla tehdyn järjestelmän yksikkötestaaminen
Työn ohjaaja: Pertti Heikkilä
Työn valmistumislukukausi ja -vuosi: kevät 2018 Sivumäärä: 35 + 6

Tämä opinnäytetyö on laadittu tilaajayritys Mekiwi Oy:n tarpeisiin ja siinä tutkitaan yksikkötestejä sekä niiden merkitystä ja hyötyjä MEAN-pinolla tehdyssä järjestelmässä. Tarve opinnäytetyölle ilmeni, kun sen kohteena olevan järjestelmän testausta haluttiin helpottaa ja automatisoida. Tavoitteena oli tuottaa yritykselle tietoa yksikkötestauksesta ja yksikkötestien rakentamisesta sekä suunnitella ja toteuttaa yksikkötestit kattamaan järjestelmän koko lähdekoodi.

Alusta asti oli selvää, että tutkimusta tarvittaisiin varsin vähän, sillä työn painopiste on toiminnallisuudessa. Lähteiden sisältö onkin suurimmaksi osaksi teoriapohjaa, jolla avataan tarvittavat käsitteet ja teknologiat itse toteutusosaa varten. Käytetyistä lähteistä suurin osa on kunkin käsiteltävän asian kotisivuja.

Opinnäytetyön tuloksena on kaksi sarjaa yksikkötestejä, jotka testaavat järjestelmän asiakassovelluksen käyttäjiä koskevia tietokantatoimintoja ja HTTP-rajapintoja. Lisäksi sen tuloksena saatiin tietoa tarvittavista toimenpiteistä yksikkötestien tekemisestä kattamaan koko järjestelmän koodi. Pelkkä testien kirjoittaminen ei tule riittämään, sillä sekä järjestelmän palvelin- että asiakassovelluksessa on koodia, jonka yksikkötestaaminen ilman koodimuutoksia on vähintäänkin epäluotettavaa ellei jopa mahdotonta.

Asiasanat: JavaScript, ohjelmointi, testaus

ABSTRACT

Oulu University of Applied Sciences
Bachelor of Computer Science and Engineering, Software Engineering

---

Author: Ilkka Koiste
Title of thesis: Unit testing a software built with the MEAN stack
Supervisor: Pertti Heikkilä
Term and year when the thesis was submitted: spring 2018 Number of pages: 35 + 6

This thesis was made for a company called Mekiwi Oy. The subject is to examine the purpose and benefits of unit tests in a software built with the MEAN stack. The need for this thesis arose when the company wanted to improve the testing of a software and to make it more automatic. The goal was to provide information to the company about unit testing and to design and produce unit tests to cover the whole of the software's codebase.

It was clear right from the start that the need for research would be limited because the main focus would be on the functional side. The majority of the source material used is information about the concepts and technologies described either in the theory section or in the implementation section. The vast majority of the references are links to a homepage of the subject in question.

The end result was two sets of unit tests: one testing the database functionalities regarding users and one testing the HTTP application programming interfaces regarding users. Information on creating unit tests to cover the entire codebase was also uncovered. Mere writing the unit testing will not be sufficient because both the client and the server have code that will need refactoring to make unit testing it possible.

Keywords: JavaScript, programming, testing

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön aiheena on yksikkötestit sekä niiden merkitys ja hyödyt MEAN-pinolla tehdyssä järjestelmässä. Teoriaosio tulee avaamaan käsitteitä MEAN-pino ja yksikkötesti sekä niihin liittyviä oheiskäsitteitä ja -teknologioita. Lisäksi ennen toteutusosiota kerrotaan kohdejärjestelmän nykytilasta ja historiasta. Toteutusosio tulee sisältämään kaksi sarjaa yksikkötestejä: käyttäjiä koskevien tietokantatoimintojen sekä HTTP-rajapintojen testaaminen. Testit ovat osittain esimerkinomaisia.

Idea aiheeseen syntyi tilaajayritys Mekiwi Oy:n tarpeesta parantaa järjestelmän testausta ja lisätä testiautomaatiota. Aihe oli kiehtova, sillä yksikkötestien väitetään toimivan koodin dokumentaationa ja parantavan koodin teknistä rakennetta. Isommassa mittakaavassa tämän opinnäytetyön toivotaan toimivan ikään kuin ohjenuorana tilaajayrityksen ohjelmistotestauksessa vähintään sillä tavalla, että se opettaa tekijällensä yksikkötestien saloja, merkitystä ja hyötyjä.

Opinnäytetyö on pyritty kirjoittamaan siten, että se olisi mahdollisimman helppolukuinen ja ymmärrettävä myös niille, jotka eivät ohjelmistotekniikkaa ymmärrä tai eivät ole tekemisissä suoraan ohjelmakoodin kanssa. Opinnäytetyö pyrkii selittämään auki mahdollisimman paljon teoriakäsitteitä.

2 MEAN-PINO

MEAN-pino on neljän JavaScript-pohjaisen teknologian muodostama rypäs, joka saa nimensä MongoDB:n, Expressin, AngularJS:n ja Node.js:n ensimmäisistä kirjaimista (MongoDB 2013, viitattu 26.2.2018). JavaScript puolestaan on ympäristostä riippuen joko tulkattava tai ajon aikana käännettävä ohjelmointikieli. Vaikka se tunnetaan parhaiten web-sivujen skriptikielenä, myös monet Internet-selaimen ulkopuoliset ympäristöt käyttävät sitä, muun muassa MEAN-pinosta tuttu Node.js. (Mozilla 2018, viitattu 1.3.2018.)

MEAN-pinon käyttämisessä on JavaScript-pohjaisuuden ansiosta selkeitä etuja, kuten hyvä suorituskyky, kehittäjien tuottavuus, virheiden etsinnän suhteellinen helppous ja tietokannan hallinnoinnin yhdenmukaisuus. Kaikkiin edellä mainittuihin asioihin liittyy tavalla tai toisella se, että kaikki käyttävät samaa kieltä sekä samanlaista tiedontallennusformaattia ja tiedonkäsittelyformaattia nimeltään JavaScript Object Notation. (MongoDB 2013, viitattu 26.2.2018.) JavaScript Object Notation eli JSON on kevyt tiedonvaihtoformaatti, joka on ihmisille helppoa lukea ja kirjoittaa sekä tietokoneiden parsia ja luoda (JSON, viitattu 1.3.2018). Se on syntaksiltaan kevyt ja yksinkertainen.

Hyvä suorituskyky selittyy muiden muassa sillä, että MongoDB-tietokantaan tallennetaan tieto käyttäen JSON-formaatin kaltaista Binary JSON -formaattia eli BSON-formaattia. Samaa JSON-formaattia käytetään myös tiedon kyselyyn tietokannasta sekä lähettämiseen ja vastaanottamiseen tietokannasta. MEAN-pinostakin tuttu AngularJS käyttää tuota samaa JSON-formaattia tiedonkäsittelyyn. (MongoDB 2013, viitattu 26.2.2018.)

Kehittäjien hyvään tuottavuuteen liittyvät hyvin pitkälti samat asiat kuin hyvään suorituskykyyn, mutta hieman eri näkökulmasta katsottuna. Saman formaatin ja kielen käyttäminen rajaa opeteltavien parhaiden tapojen ja käytäntöjen määrän yhteen. Samalla kynnys siirtyä esimerkiksi palvelinpuolen ohjelmoimisesta asiakassovellusohjelmoijaksi madaltuu, koska kieli ja tavat ovat samat. (MongoDB 2013, viitattu 26.2.2018.)

Koska tieto on samassa formaatissa joka vaiheessa ja paikassa, virheiden etsintä on tavallista helpompaa ja tietokannan hallinnointi on yksinkertaista (MongoDB 2013, viitattu 26.2.2018). Idea

kummankin väitteen takana on, että ohjelmoijan ei tarvitse etsiä vastaavuuksia tiedossa palvelimen ja asiakkaan välillä, koska tietorakenne on molemmissa sama.

2.1 MongoDB

MongoDB on niin kutsuttu NoSQL-tietokanta (MongoDB 2013, viitattu 26.2.2018). Sen hyödyt relaatiotietokantaan verrattuna ovat parempi skaalautuvuus ja parempi suorituskyky. Tämän lisäksi se vastaa paremmin nykypäivän vaatimuksiin: suuri määrä nopeasti muuttuvaa joko täysin jäsentynyttä, osittain jäsentynyttä tai ei ollenkaan jäsentynyttä tietoa; ketterän kehityksen aiheuttamat tiheät ja isotkin muutokset tietorakenteisiin; helppokäyttöinen ja mukautuva olio-ohjelmointi sekä maantieteellisesti hajautettu arkkitehtuuri. (NoSQL 2018, viitattu 27.2.2018.) NoSQL mahdollistaa tiedontallennuksen ohjenuorana käytettävän skeeman määrittelemisen kokonaisuudessaan ohjelmakoodissa. Toisin sanoen MongoDB ei välitä dokumenttien rakenteesta, kunhan ne noudattavat JSON-formaattia. (MongoDB 2013, viitattu 26.2.2018.)

MongoDB:n kanssa voidaan käyttää erillistä kirjastoa, joka antaa lisäominaisuuksia skeemapohjaiseen tiedontallennukseen. Yksi tällainen kirjasto on Mongoose. Se on MongoDB:n ja Node.js:n väliin tuleva kerros, joka tarjoaa sisäänrakennettuna tiedon validoinnin, kyselyiden rakentamisen ja koukkuja niin kutsuttuun business-logiikkakerrokseen (Mongoose 2011, viitattu 14.4.2018). Kyselyiden rakentaminen onnistuu tietysti myös ilman Mongoose-kirjastoa.

MongoDB sisältää MySQL-tietokannan tapaan komentokehotepohjaisen sovelluksen, jolla voidaan hoitaa tietokantahallinnollisia oikeuksia, kuten antaa ja poistaa käyttöoikeuksia kantoihin. Sen avulla voidaan myös suorittaa tietokantakyselyitä.

2.2 Express

Express on minimaalinen ja joustava Node.js-web-sovellusviitekehys, jossa on vankka lista ominaisuuksia web- ja mobiilisovelluksille. Express on ohut kerros perustavanlaatuisia web-sovellusominaisuuksia, jotka eivät piilota Node.js:n ominaisuuksia. Monet suositut sovellusviitekehykset perustuvat Expressiin. (ExpressJS 2017, viitattu 28.2.2018.)

Expressin filosofia on tarjota pieni ja vankka työkalupaketti HTTP-palvelimien käyttöön, mikä tekee siitä hyvän ratkaisun muiden muassa yhden sivun sovelluksiin, web-sivuihin ja julkisiin HTTP-rajapintoihin. Express ei pakota käyttämään mitään tiettyä object relational mapping -työkalua tai sapluunamoottoria. (ExpressJS 2018, viitattu 9.4.2018.) Object relational mapping eli ORM on mekanismi, joka mahdollistaa olioiden käsittelyn sekä manipuloinnin ilman tarvetta välittää sen suhteesta tietolähteeseensä (Rouse 2008, viitattu 9.4.2018). Sapluunamoottori eli template engine mahdollistaa staattisten sapluunatiedostojen käyttämisen sovelluksissa (Express Guide 2017, viitattu 9.4.2018). Express-sovelluksen voi tehdä myös käyttämättä ORM-työkalua ja sapluunamoottoria.

2.3 AngularJS

AngularJS mahdollistaa web-sovellusten rakentamisen kuin käytössä olisi älykkäämpi Internet-selain (AngularJS-github 2018, viitattu 28.2.2018). AngularJS on Model View Controller -mallin mukainen sovellusviitekehys (AngularJS 2018, viitattu 14.4.2018). Model View Controller eli MVC on malli, joka jakaa sovelluksen huolenaiheet kolmeen osaan, jotka ovat malli, näkymä ja ohjain. Ensin mainittu malli edustaa tiedon sisältävää oliota, joka voi sisältää tiedonpäivittämiseen tarvittavan logiikan. Näkymä puolestaan visualisoi ja näyttää mallin sisältämän tiedon. Ohjaimen tehtävä on hallinnoida tiedonkulkua malliin ja päivittää näkymän aina tiedon muuttuessa. (Tutorialspoint 2018, viitattu 14.4.2018.)

AngularJS:n keskeisiin käsitteisiin ja ominaisuuksiin kuuluvat muiden muassa tiedon sitominen, tavallinen JavaScript, uudelleenkäytettävät komponentit ja testattavuus. Tiedon sitominen tarkoittaa automaattista tapaa päivittää näkymä mallin muuttuessa ja päinvastoin. (AngularJS 2018, viitattu 14.4.2018.) Tarkemmin sanottuna AngularJS tukee kaksisuuntaista tiedonsidontaa. Näkymät rakentuvat HTML-merkintäkielestä, jonka syntaksin laajentamisen AngularJS mahdollistaa. (AngularJS-github 2018, viitattu 28.2.2018.) HTML-elementeille voi esimerkiksi antaa uusia attribuutteja. Myös kokonaan uusien HTML-elementtityyppien tekeminen on mahdollista.

AngularJS:n ohjaimet ja mallit ovat tavallista JavaScriptia, mikä tarkoittaa sitä, ettei niitä tarvitse periyttää mistään tietotyypistä. Malli on suoraan JavaScript-olio, minkä ansiosta se on helppoa testata, ylläpitää ja uudelleenkäyttää. (AngularJS 2018, viitattu 14.4.2018.) Web-sovelluksen hyvän teknisen rakenteen varmistamiseksi ja testauksen helpottamiseksi AngularJS hoitaa sovelluksen

tarvitsemien kirjastojen lataamisen (AngularJS-github 2018, viitattu 28.2.2018). Sovellukseen tulee toki olla kirjattuna tarvittavat kirjastot.

AngularJS:n uudemmat versiot tunnetaan nimellä Angular. Moni asia on muuttunut, mutta vaikka päivittäminen on työläs prosessi, niin sen voi tehdä pienissä osissa, mikä on yksi avaintekijöistä onnistuneessa päivityksessä. (Angular 2018, viitattu 14.4.2018.)

2.4 Node.js

Node.js on Google Chromen V8 JavaScript -moottoria käyttävä ajoympäristö palvelinkoodin ajamiseen. Se käyttää tapahtumapohjaista estotonta syöttö-tuloste-mallia, mikä tekee siitä kevyen ja tehokkaan. Sen pakettiekosysteemi Node Package Manager eli npm on maailman suurin avoimen lähdekoodin kirjastoekosysteemi maailmassa. (Node.js, viitattu 28.2.2018.) Se on tarkoitettu JavaScript-koodille ja se on ilmainen. Sille on tarjolla myös kaksi kuukausittaista maksulisenssiä, joista halvempi mahdollistaa yksityiset paketit ja kalliimpi on tarkoitettu yrityskäyttöön. (npm, viitattu 9.4.2018.)

Iso osa MEAN-pinossa käytettävistä JavaScript-kirjastoista löytyy npm-rekisteristä, josta niiden asentaminen on helppoa. Asennus tapahtuu kaikessa yksinkertaisuudessaan komennolla npm install paketin nimi. Tuolle komennolle voidaan antaa myös lisäoptioita kuten --save-dev, joka asentaa kirjaston vain kehitysympäristöön. Kirjastot asentuvat oletusarvoisesti ohjelmistokansion alikansioon node_modules, jonka npm osaa itse luoda.

Kun npm:n kautta järjestelmään on asennettu ensimmäinen mikä tahansa kirjasto, ohjelmistokansion juureen luodaan package.json-tiedosto. Se on JSON-formaattia noudattava tiedosto, joka sisältää paljon tietoa kuten ohjelmiston nimen, kuvauksen, versionumeron ja ohjelmiston käynnistystiedoston. Tiedostoon tallentuu myös lista kaikista ohjelmistoon asennetuista kirjastoista. Tuon tiedoston avulla npm osaa asentaa tarvittaessa kaikki listatut kirjastot yhdellä komennolla: npm install. Lisäksi package.json-tiedostoon voidaan lisätä skriptejä, joita tarvitaan esimerkiksi ohjelmiston kehitystyössä tai yksikkötestaamisessa.

3 YKSIKKÖTESTAUS

Ohjelmistokehityksen tärkeä ja olennainen osa on testaus, jonka avulla ohjelmistokehittäjät saavat tietoa ohjelmiston laadusta ja löytävät ohjelmistosta ongelmakohtia. Testaus voidaan jakaa erilaisiin vaiheisiin tai menetelmiin kuten yksikkötestaukseen, komponenttitestaukseen, integraatiotestaukseen ja järjestelmätestaukseen (Pan 1999, viitattu 28.2.2018). Tässä opinnäytetyössä käsitellään yksikkötestausta.

Yksikkötestaus tarkoittaa lähdekoodin yksittäisten osien testausta (AngularJS Developer Guide 2018, viitattu 12.3.2018). Yksikkötestauksen tarkoitus on validoida, että jokainen koodiyksikkö toimii, kuten se on suunniteltu toimimaan. Koodiyksikkö on ohjelmiston on pienin osa, joka voidaan testata. (Software Testing Fundamentals 2018, viitattu 16.4.2018.)

Yksikkötestauksen keskiössä ovat yksikkötestit, jotka perinteisesti määritellään koodinpätkiksi, jotka kutsuvat toisaalla olevaa ohjelmakoodia tarkistaakseen oman ennaltaodotetunlaisen toimintansa (Saleh 2013, 24). Yksikkötestillä testataan koodiyksiköitä, ja sen avulla saadaan vastaus esimerkiksi kysymyksiin: ”Ajattelinko tämän ohjelmalogiikan oikein?” ja ”Lajitteleeko tämä lajittelufunktio listan sisällön oikeaan järjestykseen?”. Muiden muassa näiden kysymysten kannalta keskeisessä osassa on käsite ongelmien eriyttäminen, joka tarkoittaa sitä, että testattava koodi kyetään erottamaan muusta koodista. Yksikkötestaamisen kannalta on nimittäin hankaloittava tekijä, jos esimerkiksi lajittelufunktiota testattaessa tieto pitää hakea palvelimelta eikä tietoa tai itse palvelinkutsua kyetä simuloimaan. (AngularJS Developer Guide 2018, viitattu 12.3.2018.) Tällöin ohjelmakoodin epähaluttu toiminnallisuus tai ulosanti voi johtua muusta koodista kuin itse testaamisen alla olleesta koodista, jolloin testi ei enää ole yksikkötesti.

Yksikkötestit ovat ohjelmistotestaamisen ensimmäinen taso ja ne suoritetaan ennen integraatiotestaamista. Yleensä yksikkötestaamista suorittavat ohjelmistokehittäjät itse tai heidän vertaisensa. Joskus harvoin yksikkötestausta voivat suorittaa myös itsenäiset ohjelmistokehittäjät. (Software Testing Fundamentals 2018, viitattu 16.4.2018.)

JavaScript-pohjaisiin kirjastoihin ja sovellusviitekehyksiin on tarjolla lukuisia kirjastoja, jotka joko ylipäätään mahdollistavat sen yksikkötestaamisen tai voivat tapauskohtaisesti olla korvaamattoman arvokkaita aputyökaluja. Hyvä esimerkki on HTTP-pyynnöt abstraktoiva ja niitä

emuloiva kirjasto supertest. Ilman sellaista kirjastoa HTTP-pyyntöjen testaaminen olisi epäluotettavaa, sillä oikea HTTP-pyyntö voi epäonnistua monesta syystä, joilla ei ole mitään tekemistä testauksen kohteena olevan ohjelmakoodin kanssa eivätkä siten välttämättä ole testausta suorittavan ohjelmoijan estettävissä.

3.1 Merkitys

Yksikkötestaus laiminlyödään usein, mutta se on itseasiassa testaamisen tärkein taso (Software Testing Fundamentals 2018, viitattu 16.4.2018). Ohjelmakoodissa olevien virheiden etsiminen ilman yksikkötestausta on täysin manuaalista, mikä tarkoittaa vähänkään isommassa ohjelmistossa paljon työtä. Kaiken lisäksi virheenkorjauksen jälkeen ei voida varmistua saman virheen toistumattomuudesta ohjelmistokehityksen myöhäisemmässä vaiheessa. (Saleh 2013, 26.) Yksikkötestauksessa tulisi pyrkiä mahdollisimman suureen prosentuaaliseen testikattavuuteen.

Mitä aiemmin koodin mahdolliset viat löytyvät, sitä nopeampi ne on korjata ja sitä vähemmän korjaaminen maksaa. Ajallisten ja rahallisten kustannusten lisäksi voi tulla aineettomia vahinkoja ja tuhoa kuten kolhuja maineeseen ja nöyryytystä. Nämä voidaan nähdä liittyvän ohjelmiston laatuun, jota yksikkötestaus parantaa itse testaamisen luotettavuuden ohella. (Software Testing Fundamentals 2018, viitattu 16.4.2018.)

3.2 Hyödyt

Yksikkötestauksen lukuisten hyötyjen joukosta merkittävimmäksi nousee toimenpiteen kertaluontoisuus. Yksikkötestin luomisen jälkeen sitä ei välttämättä tarvitse muokata, ellei sen testaama yksikkö muutu merkittävästi. Koska testit suorittaa ihmisen sijaan tietokone, testin käyttötapausten kattavuus nousee yhdestä eksponentiaaliseen määrään, mikä tuo merkittäviä aikasäästöjä jo lyhyellä aikavälillä. Vaikutus korostuu koodiyksiköissä, jotka vastaanottavan tietonsa epäsäännöllisten käyttäjätoimintojen kautta tulleista tietolähteistä kuten lomakkeista. (Korjus 2015, 12.)

Yksikkötestaus lisää varmuutta koodimuutosten tekemiseen ja koodin ylläpitoon liittyvien toimenpiteiden suorittamiseen. Jos muokkauksen alla olevassa koodista on olemassa hyvät yksikkötestit ja ne ajetaan aina koodin muuttuessa, on helppo viipymättä havaita mahdolliset

koodimuutoksen aiheuttamat viat. Lisäksi koodimuutoksella voi olla tahattomia sivuvaikutuksia, joita yksikkötestaaminen luonteensa vuoksi jo itsessään vähentää. (Software Testing Fundamentals 2018, viitattu 16.4.2018.)

Yksikkötestaaminen säästää aikaa, koska kehitystyöstä tulee nopeampaa yksikkötestien myötä, sillä ilman niitä kehittäjät suorittavat vain satunnaisia manuaalisia testejä, jotka eivät takaa ohjelman halutunlaista toimintaa. Vaikka yksikkötestien kirjoittamiseen menee aikaa, niiden kirjoittamiseen kuluvaa aikaa kompensoi testaukseen tarvittavan ajan pienentyminen. Lisäksi koko ohjelmiston kannalta katsottuna mahdollisten vikojen löytyminen yksikkötestauksessa järjestelmätestauksen sijaan on valtavan paljon halvempaa ja nopeampia korjata. (Software Testing Fundamentals 2018, viitattu 16.4.2018.)

Yksikkötestaus toimii dokumentaationa, sillä asiallisesti toteutetusta yksikkötestistä on selkeästi nähtävissä kolme yksikön toimintaan liittyvää avainelementtiä, jotka ovat syöte, toimenpide ja odotettu lopputulos (Software Testing Fundamentals 2018, viitattu 16.4.2018). Näiden avulla voidaan hyvin pitkälle päätellä ja sisäistää kyseisen yksikön käyttötarkoitus ja -tapa. Voidaan myös perustellusti väittää, että yksikkötestaaminen auttaa ohjelmoijaa kirjoittamaan paremmaa koodia, sillä se vaatii toimiakseen koodin olevan moduularista (Software Testing Fundamentals 2018, viitattu 16.4.2018).

Vaikka yksikkötestaamisesta on paljon hyötyjä, se ei yksinään ole kaiken kattavaa. Se ei löydä esimerkiksi vikaa, joka ilmenee useammasta koodiyksiköstä koostuvien moduulien yhteen liittämisestä. Sen lisäksi tarvitaan siis muitakin testausmetodeja.

4 KOHDEJÄRJESTELMÄ

Tämän opinnäytetyön kohdejärjestelmä on MEAN-pinolla rakennettu eräänlainen kevyt toiminnanohjausjärjestelmä. MEAN-pino valittiin toteutustekniikaksi, koska haluttiin moderni, mutta paljon käytetty tekniikka, jolla on ympärillään iso ja aktiivinen kehitysyhteisö. Lisäksi tämän opinnäytetyön tilaajayrityksellä oli entuudestaan kokemuksia kaikista pinon tekniikoista.

Järjestelmä siirtyi pilottivaiheeseen pitkällisen prosessin jälkeen elokuussa 2017. Noin seuraavan kuukauden aikana järjestelmästä korjattiin suurimmat ongelmat ja tämän opinnäytetyön julkaisuhetkellä se on tuotantokäytössä. Järjestelmään on annettu testitunnuksia eri tahoille ja sitä on myös käytetty pohjana toisessa tuotekehitysprojektissa.

Järjestelmän ominaisuuksiin kuuluvat käyttäjien, yritysten, toimipisteiden, ryhmien, ruokalistojen, ruokien, raaka-aineiden, tilausten ja laskujen hallinta. Hallinnalla tarkoitetaan tässä tapauksessa kohteena olevan asian lisäämistä, muokkaamista tai poistamista. Näiden toimintojen lisäksi käyttäjille on mahdollista antaa eritasoisia käyttöoikeuksia yritys-, toimipiste- ja ryhmäkohtaisesti. Järjestelmänlaajuisia asioita koskeviin toimintoihin ei tavallinen järjestelmänvalvoja kykene antamaan oikeuksia.

Järjestelmässä on kaksi käyttöoikeustasoa: järjestelmänvalvoja ja tavallinen käyttäjä. Järjestelmänvalvoja voi suorittaa kaikki järjestelmän sisältämät toiminnot, mutta tavallinen käyttäjä voi vain luoda, muokata ja poistaa ruokatilauksia, nähdä ja tulostaa ruokalistan, tarkastella koontia ruokatilauksistaan sekä tarkastella saamiaan laskuja. Listattujen ominaisuuksien olemassaolo ja laajuus riippuvat käyttäjän oikeuksista ryhmiin, toimipisteisiin ja yrityksiin.

4.1 Asiakassovellus

Järjestelmän asiakassovellus käyttöliittymineen on toteutettu AngularJS-kirjastolla. Järjestelmän kehittämisen alkamisen aikoihin AngularJS:n toinen versio oli jo beta-vaiheessa, mutta sitä ei valittu toteutustekniikaksi kahdesta syystä. Ensinnäkin sen ensimmäinen versio oli tilaajayritykselle entuudestaan tuttu, joten apua kehityksessä tuleviin mahdollisiin ongelmiin oli saatavissa lähellä.

Toisekseen järjestelmän tarpeisiin alun perin valittu kalenterikirjasto oli saatavilla vain ensimmäiselle versiolle.

Sovelluksen ulkoasu on suurimmalta osin Bootstrap-kirjaston käyttöliittymäohjeiden mukainen. Bootstrap on avoimen lähdekoodin kirjasto, jolla kehitetään käyttämällä HTML-merkintäkieltä, CSS-tyyliohjetta ja JavaScript-ohjelmointikieleltä responsiivisia ja mobiililähtöisiä web-tuotteita (Bootstrap, viitattu 4.4.2018). Järjestelmän rakentamisen aloittamisen aikoihin kirjaston neljäs versio oli alpha-vaiheessa, mutta sen sijasta päädyttiin käyttämään versiota 3.3.7. Tärkein syy tälle päätökselle oli, että edellä mainitulle kolmannelle versiolle löytyi valmiit AngularJS-direktiivit, joiden avulla ne saatiin noudattamaan AngularJS:n toimintatapoja ja käytänteitä varsin helposti. Toinen syy oli potentiaaliset ongelmakohdat alfaversion kanssa ja sen mahdollinen keskeneräisyys.

Asiakassovelluksen päivittäminen käyttämään AngularJS:n uudempaa versiota eli Angularia tulee olemaan ajankohtainen lähikuukausina tai -vuosina. AngularJS lakkaa olemasta aktiivisen kehitystyön alainen 1.7.2018 alkaen ja siirtyy kolme vuotta kestävään pitkäikäistukivaiheeseen (Darwin 2018, viitattu 23.4.2018). Tämä tarkoittaa käytännössä sitä, että uusia ominaisuuksia ei enää tule, mikä tarkoittanee enenevissä määrin sitä, että kolmannen osapuolen kirjastojen kehitys siirtyy entistä enemmän sovellusviitekehyksen uudempiin versioihin. Tämän opinnäytetyön tekemisen aikana ainakin yhden järjestelmän käyttämän AngularJS-kirjaston aktiivinen kehitys on siirtynyt suosimaan sen Angular-versiota.

4.2 Palvelinsovellus ja järjestelmän sisäiset web-rajapinnat

Palvelinsovellus toteutettiin Node.js-sovelluksena hyödyntämällä lukuisaa määrää kirjastoja, joista maininnanarvoisimmat lienevät Express ja Mongoose. Näistä kahdesta sovelluksen teknisessä rakenteessa näkyy middleware-pohjaisuutensa ansiosta enemmän Express. Rajapinta koostuu sen kuuntelemasta reitistä ja sen käsittelykoodista, jota ennen ja jonka jälkeen voidaan käyttää middleware-funktioita. Hyvä esimerkki, jota tämä järjestelmä käyttää, on autentikaatio. Se on liitetty suurimpaan osaan järjestelmän rajapinnoista ja suoritetaan aina ennen varsinaista käsittelykoodia. Tällä tavalla saadaan kätevästi kootusti yhdessä paikassa hoidettua käyttäjän autentikaation tarkistus.

Varsinaisesta käyttäjäautentikaatiosta vastaa kirjasto nimeltään passport. Se on yksinkertainen, huomaamaton, erittäin joustava, modulaarinen ja se voidaan upottaa mihin tahansa Express-pohjaiseen web-sovellukseen (Passport, viitattu 14.4.2018). Autentikaation todentamiseen järjestelmä käyttää JSON Web Token -teknologiaa. Se on avoin RFC 7519 -standardi, jonka avulla kaksi osa puolta voivat turvallisesti vaihtaa vaateita (JSON Web Token, viitattu 14.4.2018). Sen käyttöönotosta huolehtii kirjasto nimeltään passport-jwt, joka linkittää passportin ja JSON Web Tokenin eli JWT:n sekä dekoodaa, todentaa ja luo ne.

Mongoosen päävastuualueita järjestelmässä on kolme: tietokantakokoelmien malleina toimiminen ja niille lisäominaisuuksien antaminen, tietokantakyselyiden rakentaminen sekä rajapinnoissa tietokantaolioiden käyttämisen tehostaminen ja helpottaminen. Tietokantamalleista nähdään kätevästi, mitä tietoa ja missä muodossa kukin tietokannankokoelma sisältää. Mallin lisäominaisuuksista on käytössä esimerkiksi validointi, jonka avulla saadaan esimerkiksi tuotteen hinta pakotettua positiiviseksi tai sähköpostiosoite noudattamaan tarkkaa regular expressionin määräämää muotoa. Regular expression on erikoinen tekstipätkä, jolla kuvataan hakukaava (Regular-Expressions.info 2016, viitattu 14.4.2018).

Tietokantakyselyitä Mongoose helpottaa muiden muassa tarjoamalla funktion populate, jonka avulla mallin olion sisältämä toisen mallin olio voidaan kyselyvaiheessa räjäyttää auki sisältäämään muutakin tietoa kuin vain linkitykseen käytetty _id-kenttä. Samalla Mongoose sisällyttää kannasta haettuihin olioihin esimerkiksi tallennusfunktion.

4.3 Tietokanta

MEAN-pinon mukaisesti järjestelmän tietokantana on MongoDB-tietokanta. Järjestelmässä se tallentaa tiedon yhdeksään eri kokoelmaan, jotka on tarpeen mukaan linkitetty toisiinsa palvelinsovelluksen tasolla. Linkitys tehdään skeemoissa ja sen täytäntöönpanosta vastaa Mongoose. Tietokantaan tullaan todennäköisesti luomaan indeksejä, kun ajan mittaan saadaan ensin riittävästi pohjatietoa.

Järjestelmän käyttämä MongoDB-asennuksen tietokantapääsy on suojattu autentikaatiolla, mikä ei ainakaan tämän järjestelmän kehityksen alkuvaiheessa ollut MongoDB:n oletuskäytäntö. Suojaus koskee kaikkea pääsyä eli sekä järjestelmän tekemiä pyyntöjä että MongoDB-shellin pyyntöjä.

5 TOTEUTUS

Tämän opinnäytetyön kohteena olevan järjestelmän yksikkötestaamisen mahdollistamiseksi tuli ensimmäiseksi tehdä tarvittavat asennukset ja konfiguraatiot. Niiden jälkeen vuorossa on yksikkötestien suunnittelu, jonka jälkeen tehdään esivalmisteluja testejä varten. Lopuksi kirjoitetaan itse yksikkötestit.

5.1 Testiympäristön asennus ja konfigurointi

Ensimmäinen toimenpide testiympäristön käyttökuntoon saamiseksi on varmistaa, että npm on asennettu, koska sen avulla tullaan asentamaan yksikkötestauksessa tarvittavat kirjastot. Asennus tapahtuu käyttöjärjestelmästä riippuen hieman eri tavalla. Tämän opinnäytetyön tekemiseen käytetyn Applen MAC-tietokoneen kohdalla voidaan turvautua esimerkiksi kolmannen osapuolen Homebrew-paketinhallintajärjestelmään, jonka avulla npm asennetaan syöttämällä terminaaliin komento: brew install node.

Toinen toimenpide on asentaa kirjasto, jonka avulla ylipäätään voidaan yksikkötestata Node.js-sovellusta. Tarjolla on useita eri vaihtoehtoja. Tämän opinnäytetyön käyttöön valittiin kirjasto nimeltään Mocha, koska se on suosittu ja se oli ainakin pieniltä osin entuudestaan tuttu. Mocha on monipuolinen JavaScript-sovellusviitekehys Node.js- ja selainalustoille, joka tekee asynkroonisesta testaamisesta hauskaa ja helppoa (Mocha 2018, viitattu 13.4.2018). Se asennetaan ajamalla komento: npm install mocha --save-dev. Vaikka npm-paketinhallintajärjestelmää ei ole erikseen asennettu, se on asentunut node-paketin mukana. Huomionarvoista käytetyssä komennossa on niin kutsuttu kytkin --save-dev. Se listaa asennetun moduulin package.json-tiedostossa oletusarvoisen dependencies-kentän sijasta devDependencies-kentän alle, mikä tarkoittaa, että moduuli asentuu ainoastaan kehityspuolelle.

Kolmas vaihe on asentaa Mochan lisäksi kolme kappaletta kirjastoja: Chai, Sinon ja Request. Ne eivät varsinaisesti ole välttämättömiä yksikkötestaamisen kannalta, mutta niiden käyttäminen katsottiin tämän opinnäytetyön tapauksessa hyödylliseksi. Chai on niin kutsuttu assertion-kirjasto Nodelle, jota voi käyttää minkä tahansa sovellustestiviitekehyksen kanssa (Chai Assertion Library, viitattu 5.5.2018). Myös Sinon toimii minkä tahansa yksikkötestausviitekehyksen kanssa ja se

tarjoaa yksikkötestien tekemiseen erilaisia tapoja teeskennellä esimerkiksi funktiokutsuja. (Sinon.JS 2018, viitattu 5.5.2018). Request-kirjasto on suunniteltu helpoimmaiksi mahdolliseksi tavaksi tehdä HTTP-pyyntöjä (Request 2018, viitattu 8.5.2018). Varsinkin kirjastot Sinon ja Request ovat erittäin hyödyllisiä, sillä niiden avulla rajapinta- ja tietokantatoimintoyksikkötestit on mahdollista eristää funktioista ja toiminnoista, jotka eivät ole sen hetkisen testin alaisia. Koska käytössä on Mongoose, sinon tarvitsee liitännäiskirjaston nimeltään sinon-mongoose. Kaikki edellä mainitut kirjastot voidaan asentaa yksitellen tai yhdellä kertaa komennolla: npm install chai sinon sinon-mongoose request --save-dev.

Asennuksen neljäs vaihe on vapaaehtoinen, mutta sen tekeminen on suositeltavaa ja säästää aikaa jo pienissäkin ohjelmistoissa. Kun käytössä on Mocha, testit ajetaan oletusarvoisesti manuaalisesti suorittamalla sovelluksen juurikansiossa komento: node_modules/mocha/bin/mocha. Jos testien ajaminen halutaan automatisoida, se vaatii konfigurointia, joka tehdään sovelluksen juuressa olevaan package.json-tiedostoon. Konfiguraation sisältö riippuu muiden muassa, että onko käytössä jokin tuotantokäyttöön tarkoitettu prosessimanageri kuten PM2, kehityskäyttöön tarkoitettu prosessin monitorointityökalu kuten nodemon vai ajetaanko sovellusta suoraan Nodella. Jos käytössä on edellä mainittu nodemon, konfiguraatio voi olla esimerkiksi kuvan 1 mukainen.

```
36        "scripts": {
37            "test": "NODE_ENV=test nodemon --exec 'mocha --recursive -R min'"
38        },
```

KUVA 1. Yksikkötestauksen automatisoiva konfiguraatio

Yllä näkyvässä konfiguraation test-kentässä asetetaan Noden ympäristömuuttuja NODE_ENV asentoon test ja käsketään nodemon suorittamaan mocha siten, että myös alikansiot otetaan mukaan ja raportointi tehdään minimaalisesti. Vaikka test-kentän arvossa ei erikseen mainita mitään muutosten seuraamisesta esimerkiksi --watch-kytkimellä, skripti suoritetaan nodemonin ansiosta silti aina koodimuutosten tallentuessa.

5.2 Testien suunnittelu

Kuten on monen muunkin asian kanssa, suunnittelu on tärkeää myös hyvässä testaamisessa. Perinteisesti testaamisen suunnittelun keskiössä ovat testitapaukset, joita vasten testausta

suoritetaan. Ongelmien eriyttämisen filosofiaa noudattaen testitapauksen on tarkoitus sisältää vain yksi testattavana olevan ohjelmiston käyttötapaus. (Jorgensen 2013, 4.) Tämä on erityisen tärkeää yksikkötestauksessa, koska mahdollisten ongelmatilanteiden halutaan johtuvan vain ja ainoastaan itse testattavasta koodista.

Aivan jokaista asiaa varten ei välttämättä ole tarkoituksenmukaista tehdä testitapausta. Sen sijaan kannattaa keskittyä kirjoittamaan testitapaukset, jotka keskittyvät järjestelmän käyttäytymisen testaamiseen. (Software Testing Fundamentals 2018, viitattu 16.4.2018.) Opinnäytetyötä koskevien käytännönrajoitteiden vuoksi päätettiin valita käyttäjiä koskevien rajapintojen ja tietokantatoimintojen testaaminen.

5.3 Esivalmistelut testejä varten

Ennen testien kirjoittamista on syytä miettiä muutamaa käytännön asiaa testaamisesta kuten mitä tietokantainstanssia käytetään. Vaihtoehtoja on käytännössä kaksi. Joko käytetään kehityspuolen tietokantaa tai luodaan testejä varten oma tietokanta. Tässä opinnäytetyössä päädyttiin tekemään oma tietokanta testejä varten kahdesta syystä. Erillistä kantaa käyttäessään yksikkötestit eivät muuta kehityspuolen tietokannan sisältämää tietoa ja testien käyttämän kannan sisältämä tieto on paremmin kontrollissa. Erillisen kannan käyttämistä varten tarvitaan aputoimintoja, jotka esitellään kuvassa 2.

```
const configDB = require('../config/database');
const mongoose = require('mongoose');

before((done) => {
  mongoose.connect(`${configDB.url}_test`);
  mongoose.connection
    .once('open', () => done())
    .on('error', (err) => console.warn('Warning:', err));
});

beforeEach((done) => {
  const { users } = mongoose.connection.collections;

  users.drop(() => {
    done();
  });
});
```

KUVA 2. Tietokantaa testaavien yksikkötestien aputoiminnot

Yllä olevan kuvan kaksi ensimmäistä koodiriviä ottavat mukaan tarvittavat moduulit eli tietokantakonfiguraation ja Mongoose-kirjaston. Node.js-ympäristössä jokainen kooditiedosto suoritetaan eristyksissä muista kooditiedostoista, minkä vuoksi toisessa tiedostossa sijaitseva tieto tai keino on otettava erikseen mukaan. Tämä tapahtuu require-funktiolla. Joitakin poikkeuksia tosin on ja kuvassa näkyy niistä kaksi: funktiot before ja beforeEach.

Kuvan neljännellä rivillä esiintyy before-funktio. Se suoritetaan aina ennen testien ajamista. Tuolle funktiolle annetaan parametrina funktio, joka ottaa yhteyden käytettävään MongoDB-testitietokantaan. Rivillä viisi yritetään ottaa yhteys kantaan käyttäen tietokantakonfiguraatiossa määriteltyä polkua Rivit kuudesta kahdeksaan käsittelevät yhteydenoton onnistumisen ja epäonnistumisen. Kuvan rivillä 11 annetaan beforeEach-funktiolle parametrina funktio, jonka avulla käsketään users-kokoelmaa tyhjentämään itsensä aina ennen jokaista testiä. Käytännössä tämä hoidetaan hakemalla suoraan mongoose-oliosta users-malli, jonka drop-funktio tyhjentää mallin sisältämät tiedot.

5.4 Testien kirjoittaminen

Yleisellä tasolla testitapaukset noudattavat muotoa: testin esiehdot, testissä suoritettava toiminta ja testin odotetut lopputulokset. Testin suorittamisen jälkeen dokumentoidaan testin suoritusajankohta, kuka sen suoritti ja millainen oli sen lopputulos. Saadun lopputuloksen vastaavuutta ennaltaodotettuun lopputulokseen on myös syytä arvioida. (Jorgensen 2013, 4–5.)

Yksikkötesteillä on yhteinen tekninen perusrakenne. Ensin otetaan mukaan tarvittavat moduulit require-funktiolla, minkä jälkeen kutsutaan describe-funktiota kahdella parametrilla: string-muotoinen testin kuvaus ja callback-funktio, jonka sisällä listataan varsinaiset testit eli it-funktiot. Myös it-funktioita kutsutaan kahdella parametrilla, jotka ovat tismalleen samanlaiset kuin describe-funktiota käytettäessä.

Yksi describe-funktion callback-funktio voi siis sisältää monta testiä eli it-funktiokutsua. Se voi sisältää myös toisen describe-funktiokutsun. Tässä opinnäytetyössä rajattiin kuitenkin selvyyden vuoksi yksi describe-funktiokutsu sisältämään tasan yhden testin.

Kun testi on kirjoitettu, sen toimivuus tulee testata. Jos testi on kirjoitettu erheettömästi ja loogisesti oikein, sen kuuluisi mennä testejä ajaessa läpi. Kun testi menee läpi, sen rakennetta on suositeltavaa muuttaa väliaikaisesti siten, että testissä aiheutuu tilanne, josta sen ei kuuluiskaan mennä läpi. Tällä tavalla saadaan suurempi varmuus siihen, että testitapaus todella toimii tarkoitetulla tavalla.

Vaikka testin logiikan tarkoituksenmukainen toimiminen tarkistettaisiinkin muuttamalla väliaikaisesti sen koodia virhetilanteen aiheuttavaksi, on silti erittäin hyödyllistä tehdä erilliset testit virhetilanteiden ja muidenkin poikkeustilanteiden varalle. Hyvä esimerkki on testitapaus, jossa testataan käyttäjien hakemista tietokannasta ja tietokannassa ei olekaan yhtään käyttäjää. Toinen esimerkki voisi olla käyttäjän poistaminen kannasta käyttäen id-kenttää, jolle ei löydy vastinetta tietokannasta. Tällaiset rajattiin kuitenkin opinnäytetyöstä pois.

5.4.1 Tietokantatoiminnot

Tietokantatoimintoja testattaessa keskeisessä osassa on sinon-kirjasto, jonka avulla luodaan mock-funktioita. Mock on valemetodi, joilla on ennaltamäärätty käyttäytyminen ja ennaltamäärätyt odotukset, joiden täyttymättä jääminen estää testiä menemästä läpi (Mocks – Sinon.JS 2018, viitattu 8.5.2018). Mock-funktioiden avulla tietokantatoimintojen koodien testaavat yksikkötestit saadaan eristettyä koodeista, joihin niiden ei haluta olevan yksikkötestien luonteen mukaisesti olevan yhteydessä.

Koska ilman käyttäjiä järjestelmässä ei tapahtuisi mitään, testejä varten valittiin käyttäjiä koskevat tietokantatoiminnot. Niitä on taulukon 1 mukaisesti neljä kappaletta.

TAULUKKO 1. Järjestelmän käyttäjiä koskevat tietokantatoiminnot

Toiminto	User-mallin funktio
Hae kaikki käyttäjät	Find
Lisää käyttäjä	Save
Päivitä käyttäjä	Save
Poista käyttäjä	Delete

Taulukon yksi järjestyksen mukaisesti ensimmäisenä testivuorossa on koodi, joka luo uuden käyttäjän tietokantaan. Testissä käytetään niin kutsuttua mock-funktiota, jonka avulla testi voidaan eristää yksikkötestiksi. Samalla näemme ensimmäisen yksikkötestin. Kuvassa 3 näkyy edellä mainittu testi.

```
1   const sinon = require('sinon');
2   const chai = require('chai');
3   const { expect } = chai;
4   const mongoose = require('mongoose');
5   require('sinon-mongoose');
6
7   const User = require('../models/User');
8
9   describe('Reading records', () => {
10    it("should return all users", (done) => {
11      const UserMock = sinon.mock(User);
12      const expectedResult = { status: true, users: [] };
13
14      UserMock.expects('find').yields(null, expectedResult);
15
16      User.find((err, result) => {
17        UserMock.verify();
18        UserMock.restore();
19        expect(result.status).to.be.true;
20        done();
21      });
22    });
23  });
```

KUVA 3. Käyttäjien hakeminen tietokannasta

Testin seitsemän ensimmäistä riviä ottavat mukaan tarvittavat resurssit. Seitsemännellä rivillä mukaan otettava User-skeema löytyy liitteenä 1. Rivillä yhdeksän annetaan describe-funktiolle kaksi parametria: string-muotoinen kuvaus Reading records sekä callback-funktio, joka sisältää varsinaisen yksikkötestin. Testi luodaan rivillä 10 it-funktiolla, jolle myös annetaan kaksi parametria: string-muotoinen nimi should return all users ja callback-funktion, jossa kuvataan testin toiminnallisuus. Rivillä 11 luodaan niin kutsuttu mock-funktio käyttäen suoraan mukaan otettua User-mallia. Seuraavalla rivillä luodaan odotettu lopputulos. Rivillä 14 ohjeistetaan mock-funktiota reagoimaan ensimmäiseen vastaanottamaansa find-callback-funktioon ja käyttämään odotettua lopputulosta. Rivillä 16 suoritetaan varsinainen käyttäjien hakeminen User-mallin find-funktiolla, jolle annetaan parametrit virhe-olio ja lopputulos. Rivillä 17 verifoidaan kaikki mock-funktion

odotukset. Rivillä 18 palautetaan mock lähtötilaan. Rivillä 19 tarkistetaan, että onko lopputuloksen status arvoltaan true.

Seuraava testattava tietokantatoiminto on käyttäjän lisääminen tietokantaan. Testin rakenne poikkeaa jonkin verran käyttäjien hakemisen koodin testaavasta testistä. Rakenne on nähtävillä kuvassa 4.

```
1   const sinon = require('sinon');
2   const chai = require('chai');
3   const { expect } = chai;
4   const mongoose = require('mongoose');
5   require('sinon-mongoose');
6
7   const User = require('../models/User');
8
9   describe('Creating records', () => {
10    it('should create new post', (done) => {
11      const UserMock = sinon.mock(new User({
12        email: 't2koil00@students.oamk.fi',
13        forename: 'Ilkka',
14        surname: 'Koiste'
15      }));
16      const user = UserMock.object;
17      const expectedResult = { status: true };
18
19      UserMock.expects('save').yields(null, expectedResult);
20
21      user.save((err, result) => {
22          UserMock.verify();
23          UserMock.restore();
24          expect(result.status).to.be.true;
25          done();
26      });
27    });
28  });
```

KUVA 4. Käyttäjän lisääminen tietokantaan

Ensimmäiset seitsemän riviä ottavat testiin mukaan tarvittavat resurssit. Seuraavaksi rivillä yhdeksän testille määritellään describe-funktiolla kuvaukseksi Creating records. Edellä mainitulle funktiolle annetaan myös parametrina funktio, joka sisältää rivillä 10 it-funktion, joka on varsinaisen testi. Samalla rivillä testin nimeksi määritellään should create new post ja it-funktiolle annetaan parametrina funktio, joka sisältää varsinaisen testin toiminnallisuuden. Rivillä 11 luodaan käyttäjästä niin kutsuttu mock-funktio. Rivillä 16 luodaan mockista varsinainen käyttäjäolio, joka talletetaan kantaan rivillä 21. Sitä ennen rivillä 17 luodaan odotettu lopputulos ja rivillä 19 kerrotaan

se mock-funktiolle. Talletusfunktiolle annetaan parametrina funktio, joka verifoi lopputuloksen, palauttaa mockin lähtötilanteeseen ja päättää testin.

Seuraavaksi kirjoitetaan testi käyttäjän päivittämiseksi tietokannassa. Sen rakenne on yhtä poikkeusta lukuunottamatta identtinen käyttäjän luomisen testaavan yksikkötestin kanssa. Testi on nähtävillä kuvassa 5.

```
1   const sinon = require('sinon');
2   const chai = require('chai');
3   const { expect } = chai;
4   const mongoose = require('mongoose');
5   require('sinon-mongoose');
6
7   const User = require('../models/User');
8
9   describe('Updating records', () => {
10    it("should update a user by id", (done) => {
11      const UserMock = sinon.mock(new User({
12        email: 't2koil00@students.oamk.fi',
13        forename: 'Ilkka',
14        surname: 'Koiste'
15      }));
16      const user = UserMock.object;
17      const expectedResult = { status: true };
18
19      UserMock.expects('save')
20        .withArgs({ _id: 12345 })
21        .yields(null, expectedResult);
22
23      user.save((err, result) => {
24        UserMock.verify();
25        UserMock.restore();
26        expect(result.status).to.be.true;
27        done();
28      });
29    });
30  });
```

KUVA 5. Käyttäjän päivittäminen tietokannassa

Yllä olevan kuvan koodin seitsemän ensimmäistä riviä ottavat mukaan tarvittavat asiat. Sen jälkeen rivillä yhdeksän testeille annetaan kuvaukseksi Updating records ja rivillä 10 itse testin nimeksi should update a user by id. Käyttäjän luovan yksikkötestin tapaan mock-funktio luodaan jälleen User-mallista, jolle annetaan parametrina käyttäjäolio. Ainoa eroavaisuus käyttäjän luovaan yksikkötestiin on rivillä 19 mock-funktiolle annettava oletus, jossa käytetään tällä kertaa lisäksi

funktiota withArgs. Tuon funktion tarkoituksena on varmistaa, että mock-funktiota kutsutaan _id-parametrilla, jonka arvo on 12345.

Seuraavaksi kirjoitetaan testi käyttäjän poistamiseksi tietokannasta. Vaikka testattava operaatio on hyvin erilainen kuin edellisellä kerralla, testin kuvassa 6 näkyvä rakenne on silti varsin samanlainen.

```
1   const sinon = require('sinon');
2   const chai = require('chai');
3   const { expect } = chai;
4   const mongoose = require('mongoose');
5   require('sinon-mongoose');
6
7   const User = require('../models/User');
8
9   describe('Deleting records', () => {
10    it("should delete a user by id", (done) => {
11      const UserMock = sinon.mock(User);
12      const expectedResult = { status: true };
13
14      UserMock.expects('remove')
15        .withArgs({ _id: 12345 })
16        .yields(null, expectedResult);
17
18      User.remove({ _id: 12345 }, (err, result) => {
19        UserMock.verify();
20        UserMock.restore();
21        expect(result.status).to.be.true;
22        done();
23      });
24    });
25  });
```

KUVA 6. Käyttäjän poistaminen tietokannasta

Yllä olevan kuvan koodin seitsemän ensimmäistä riviä ottavat mukaan tarvittavat asiat. Sen jälkeen testeille annetaan rivillä yhdeksän kuvaukseksi Deleting records. Itse testille annetaan rivillä 10 nimeksi should delete a user by id. Tällä kertaa mock-funktio luodaan suoraan User-mallista, koska emme ole lisäämässä tai muuttamassa käyttäjää. Käyttäjän päivittämisen testaavan yksikkötestin tapaan tälläkin kertaa käytetään withArgs-funktiota varmistamaan, että mock-funktiokutsu sisältää argumentin _id. Huomionarvoinen eroavaisuus käyttäjän päivittämisen testaavan yksikkötestin rakenteeseen on, että User-mallin metodia remove kutsuttaessa sille annetaan parametriksi sama id kuin mock-funktiolle annetulle odotukselle.

5.4.2 Rajapinnat

Rajapintoja testattaessa keskeisessä osassa on sinon-kirjaston stub-funktio, jonka avulla luodaan stub-funktioita. Stub on funktio, jolla on ennaltamäärätty käyttäytyminen ja joka tukee sen itsensä käyttäytymisen muuttamiseen tarkoitettujen metodien lisäksi täyttä testi-spy-rajapintaa (Stubs – Sinon.JS 2018, viitattu 8.5.2018). Spy puolestaan on funktio, joka tallentaa siihen kohdistuneiden kutsujensa argumentit, paluuarvon, this-muuttujan arvon sekä mahdollisen poikkeuksen. Spy-funktioita on kahdenlaisia: anonyymeja ja testattavana olevan järjestelmän sisältäviä metodeja paketoivia. (Spies – Sinon.JS 2018, viitattu 8.5.2018.) Stub-funktioiden avulla tietokantatoimintojen koodien testaavat yksikkötestit saadaan eristettyä koodeista, joihin niiden ei haluta yksikkötestien luonteen mukaisesti olevan yhteydessä.

Keskeisimmät erot tietokantatoiminnot testaavien testien ja rajapinnat testaavien testien välillä ovat se, että rajapintoja testattaessa tarvitaan request-kirjasto, mock-funktion sijasta stub-funktio ja erillinen testitiedon sisältävä json-tiedosto, joka on liitteen 2 mukainen. Tuo tiedosto sisältää muiden muassa käyttäjät ja kaikkien pyyntöjen statuskoodit ja vastausviestit. Tiedoston avulla testit eivät tarvitse yhteyttä tietokantaan. Mainittu request-kirjasto mahdollistaa pyyntöjen simuloinnin siten, että testi ei oikeasti tee pyyntöä. Koska testit ainoastaan matkivat HTTP-pyyntöjä, testejä ajaessa voi olla varma, että ne eivät epäonnistu mahdolliseen HTTP-virheeseen, mikä edesauttaa yksikkötestien filosofian toteutumista.

Testattaviksi rajapinnoiksi valittiin ne, jotka vastaavat testattuja tietokantatoimenpiteitä. Luotavat yksikkötestit rajapintojen testaamiseksi ovat listattuna taulukossa 2.

TAULUKKO 2. Järjestelmän käyttäjiä koskevat HTTP-rajapinnat

Toiminto	HTTP-metodi	Reitti
Hae kaikki käyttäjät	GET	/api/users
Lisää käyttäjä	POST	/api/users
Päivitä käyttäjä	PUT	/api/users/:id
Poista käyttäjä	DELETE	/api/users/:id

Ensimmäinen testattava rajapinta on käyttäjien hakeminen. Kuvasta 7 nähdään, että testi poikkeaa merkittävästi tietokantatesteistä, vaikka perusrakenne on edelleen sama.

```
1   const sinon = require('sinon');
2   const request = require('request');
3   const chai = require('chai');
4   const should = chai.should();
5
6   const users = require('./fixtures/users.json');
7   const configApi = require('./config/api');
8
9   describe('users service', () => {
10    beforeEach(() => {
11      this.get = sinon.stub(request, 'get');
12    });
13
14    afterEach(() => {
15      request.get.restore();
16    });
17
18    describe('GET /api/users', () => {
19      it('should return all users', (done) => {
20        this.get.yields(
21          null, users.all.success.res, JSON.stringify(users.all.success.body)
22        );
23        request.get(`${configApi.base}/api/users`, (err, res, body) => {
24          res.statusCode.should.eql(200)
25          res.headers['content-type'].should.contain('application/json');
26          body = JSON.parse(body);
27          body.status.should.eql('success');
28          body.data.length.should.eql(2);
29          body.data[0].should.include.keys('id', 'email', 'forename', 'surname');
30          body.data[0].email.should.eql('t2koil00@students.oamk.fi');
31          done();
32        });
33      });
34    });
35  });
```

KUVA 7. Rajapinta käyttäjien hakemiseen

Ensimmäiset seitsemän riviä ottavat tuttuun tapaan mukaan kaikki tarvittavat moduulit. Rivillä 11 luodaan ennen jokaista testiä stub-funktiolla get-olio. Rivillä 15 palautetaan get-pyyntö oletustilaan. Itse testi luodaan rivillä 19. Riveillä 20–22 käsketään luotua get-oliota reagoimaan ensimmäiseen vastaanottamaansa callback-funktioon. Rivillä 23 luodaan get-kutsu, ja varsinaiset testin odotukset ovat riveillä 24–30.

Seuraava testattava rajapintakoodi on käyttäjän lisääminen. Vaikka yksikkötestin perusrakenne on jälleen sama, testi poikkeaa merkittävästi tietokantatesteistä. Sen rakenne on nähtävillä kokonaisuudessaan kuvassa 8.

```
const sinon = require('sinon');
const request = require('request');
const chai = require('chai');
const should = chai.should();

const users = require('./fixtures/users.json');
const configApi = require('./config/api');

describe('users service', () => {
  beforeEach(() => {
    this.post = sinon.stub(request, 'post');
  });

  afterEach(() => {
    request.post.restore();
  });

  describe('POST /api/users', () => {
    it('should return the user that was added', (done) => {
      const options = {
        body: {
          email: 'ilkka.koiste@mekiwi.org',
          forename: 'Ilkka',
          surname: 'Koiste'
        },
        json: true,
        url: `${configApi.base}/api/users`
      };
      const obj = users.add.success;
      this.post.yields(null, obj.res, JSON.stringify(obj.body));
      request.post(options, (err, res, body) => {
        res.statusCode.should.equal(201);
        res.headers['content-type'].should.contain('application/json');
        body = JSON.parse(body);
        body.status.should.eql('success');
        body.data[0].should.include.keys('id', 'email', 'forename', 'surname');
        body.data[0].email.should.eql('ilkka.koiste@mekiwi.org');
        done();
      });
    });
  });
});
```

KUVA 8. Rajapinta käyttäjän lisäämiseen

Ensimmäiset seitsemän riviä ottavat mukaan tarvittavat moduulit. Rivillä 11 luodaan ennen jokaista testiä stub-funktiolla post-olio. Rivillä 15 jokaisen testin jälkeen palautetaan post-pyyntö oletustilaan. Itse testi luodaan rivillä 19. Riveillä 20–29 luodaan tarvittavat muuttujat. Rivillä 30 käsketään luotua post-oliota reagoimaan ensimmäiseen vastaanottamaansa callback-funktioon.

Seuraava testattava rajapinta on käyttäjien päivittäminen. Testi poikkeaa jonkin verran edellisestä. Sen rakenne näkyy kuvassa 9.

```
1   const sinon = require('sinon');
2   const request = require('request');
3   const chai = require('chai');
4   const should = chai.should();
5
6   const users = require('./fixtures/users.json');
7   const configApi = require('./config/api');
8
9   describe('users service', () => {
10    beforeEach(() => {
11      this.put = sinon.stub(request, 'put');
12    });
13
14    afterEach(() => {
15      request.put.restore();
16    });
17
18    describe('PUT /api/users/:id', () => {
19      it('should return the user that was updated', (done) => {
20        const options = {
21          body: { email: 'ilkka.koiste@mekiwi.org' },
22          json: true,
23          url: `${configApi.base}/api/users/1`
24        };
25        const obj = users.update.success;
26        this.put.yields(null, obj.res, JSON.stringify(obj.body));
27        request.put(options, (err, res, body) => {
28          res.statusCode.should.equal(200);
29          res.headers['content-type'].should.contain('application/json');
30          body = JSON.parse(body);
31          body.status.should.eql('success');
32          body.data[0].should.include.keys('id', 'email', 'forename', 'surname');
33          body.data[0].email.should.eql('ilkka.koiste@mekiwi.org');
34          done();
35        });
36      });
37    });
38  });
```

KUVA 9. Rajapinta käyttäjän päivittämiseksi

Ensimmäiset seitsemän riviä ottavat tuttuun tapaan mukaan kaikki tarvittavat moduulit. Rivillä 11 luodaan ennen jokaista testiä stub-funktiolla put-olio. Rivillä 15 jokaisen testin jälkeen palautetaan request-instassin put-olio ennalleen. Itse testi luodaan rivillä 19 ja annetaan sille nimeksi should return the user that was updated. Riveillä 20–25 luodaan tarvittavat muuttujat. Rivillä 26 käsketään luotua put-oliota reagoimaan ensimmäiseen vastaanottamaansa callback-funktioon. Rivillä 31 luodaan put-kutsu, ja varsinaiset testit ovat riveillä 28–33.

Seuraava testattava rajapinta on käyttäjien poistaminen. Testi poikkeaa jonkin verran edellisestä. Sen rakenne näkyy kuvassa 10.

```
1   const sinon = require('sinon');
2   const request = require('request');
3   const chai = require('chai');
4   const should = chai.should();
5
6   const users = require('./fixtures/users.json');
7   const configApi = require('./config/api');
8
9   describe('users service', () => {
10    beforeEach(() => {
11      this.delete = sinon.stub(request, 'delete');
12    });
13
14    afterEach(() => {
15      request.delete.restore();
16    });
17
18    describe('DELETE /api/users/:id', () => {
19      it('should return the user that was deleted', (done) => {
20        const obj = users.delete.success;
21        this.delete.yields(null, obj.res, JSON.stringify(obj.body));
22        request.delete(`${configApi.base}/api/users/1`, (err, res, body) => {
23          res.statusCode.should.equal(200);
24          res.headers['content-type'].should.contain('application/json');
25          body = JSON.parse(body);
26          body.status.should.eql('success');
27          body.data[0].should.include.keys('id', 'email', 'forename', 'surname');
28          body.data[0].email.should.eql('t2koil00@mekiwi.org');
29          done();
30        });
31      });
32    });
33  });
```

KUVA 10. Rajapinta käyttäjän poistamiseksi

Ensimmäiset seitsemän riviä ottavat tuttuun tapaan mukaan kaikki tarvittavat moduulit. Rivillä 11 luodaan ennen jokaista testiä stub-funktiolla delete-olio. Rivillä 15 jokaisen testin jälkeen palautetaan request-instassin delete -olio ennalleen. Itse testi luodaan rivillä 19 ja annetaan sille nimeksi should return the user that was updated. Rivillä 20 luodaan tarvittava muuttuja. Rivillä 21 käsketään luotua delete-oliota reagoimaan ensimmäiseen vastaanottamaansa callback-funktioon. Rivillä 22 luodaan delete-kutsu, ja varsinaiset testit ovat riveillä 23–28.

6 POHDINTA

Tämän opinnäytetyön aiheena ja tavoitteena oli MEAN-pinolla toteutetun järjestelmän yksikkötestaaminen. Lopputuloksena oli teoriaosion lisäksi kaksi sarjaa yksikkötestejä, joista toinen testaa käyttäjiä koskevat tietokantatoiminnot ja toinen testaa HTTP-rajapinnat.

Alkuperäinen suunnitelma oli laatia yksikkötestit koko järjestelmästä, mutta tästä tavoitteesta jouduttiin luopumaan aikarajoitusten vuoksi. Tästä huolimatta esimerkiksi asiakassovellusta koskeva teoriaosuus haluttiin säilyttää, koska opinnäytetyön kohdejärjestelmä on MEAN-pinolla tehty, joten sen aukiselittäminen katsottiin hyödylliseksi. Näiden rajausten jälkeen jouduttiin myöhemmässä vaiheessa tekemään vielä toinen rajaus. Koko palvelinsovelluksen testaamisen sijaan testattaisiin käyttäjiä koskevat HTTP-rajapinnat ja tietokantatoiminnot.

Opinnäytetyötä tehdessä ehkäpä suurimmaksi hankaluudeksi osoittautui itse yksikkötestien kirjoittaminen. Ongelma oli niiden luonteen sisäistäminen: yksittäisen testin kuuluu todellakin testata vain yhtä funktiota tai metodia. Alun perin nimittäin testejä kirjoitettiin neljä kappaletta, mutta niiden huomattiin testaavan vähintään kahta eri asiaa, mikä teki niistä integraatiotestejä. Vaihtoehtoja ongelman ratkaisemiseksi oli joko muuttaa opinnäytetyön aihetta sallimaan integraatiotestit tai muuttaa testejä. Ongelma ratkaistiin pilkkomalla kukin testi kahteen osaan, jolloin lopputuloksena oli erilliset testit testatuille tietokantatoiminnoille ja HTTP-rajapinnoille.

Yksikkötestejä suunnitellessa ja rakentaessa kävi ilmi, että koko palvelinpuolen koodin kattaminen yksikkötesteillä tulisi viemään etukäteen arveltua enemmän aikaa jo pelkästään siitä syystä, että koodi vaatisi enemmän muutoksia kuin etukäteen arveltiin. Tilanne osoittautui huonommaksi asiakassovelluksen puolella. Siellä sovelluksen tekninen rakenne oli vähemmän yksikkötestiystävällinen, minkä vuoksi se rajattiin opinnäytetyön pois.

Opinnäytetyön tuloksia eli yksikkötestejä voidaan käyttää mallina palvelinpuolen koodin täydellisen yksikkötestikattavuuden rakentamisessa. Uskomus on, että ne vähintäänkin auttavat asiakassovelluksen yksikkötestien rakentamisessa, sillä niiden tekninen perusrakenne vaikuttaa olevan riittävän samanlainen kuin palvelinpuolen yksikkötesteissä.

LÄHTEET

AngularJS 2018. Viitattu 14.4.2018, https://angularjs.org/.

AngularJS Developer Guide 2018. Unit Testing. Viitattu 12.3.2018, https://docs.angularjs.org/guide/unit-testing.

AngularJS-github 2018. Github. Viitattu 28.2.2018, https://github.com/angular/angular.js.

Angular 2018. Upgrading from AngularJS to Angular. Viitattu 14.4.2018, https://angular.io/guide/upgrade.

Bootstrap. Viitattu 4.4.2018, https://getbootstrap.com/.

Chai Assertion Library. Viitattu 5.5.2018, http://www.chaijs.com/.

Darwin, Pete Bacon. Stable AngularJS and Long Term Support. Angular Blog. Viitattu 23.4.2018, https://blog.angular.io/stable-angularjs-and-long-term-support-7e077635ee9c.

ExpressJS 2017. Viitattu 28.2.2018, https://expressjs.com/.

ExpressJS 2018. Github. Viitattu 9.4.2018, https://github.com/expressjs/express.

Express Guide 2017. Using template engines with Express. Viitattu 9.4.2018, https://expressjs.com/en/guide/using-template-engines.html.

Jorgensen, Paul C. 2013. Software Testing: A Craftsman's Approach, Fourth Edition. CRC Press.

JSON. Introducing JSON. Viitattu 1.3.2018, https://www.json.org/.

JSON Web Token. Viitattu 14.4.2018, https://jwt.io/.

Korjus, Nina 2015. Testauksen perusteet. Lappeenrannan teknillinen yliopisto. PDF-dokumentti. Viitattu 14.4.2018, https://noppa.lut.fi/noppa/opintojakso/ct60a4150/materiaali/ohjelmisto-testauksen_perusteet_-manuaali.pdf.

Mocha 2018. Viitattu 13.4.2018, https://mochajs.org/.

Mocks – Sinon.JS 2018. Viitattu 8.5.2018, http://sinonjs.org/releases/v1.17.7/mocks/.

MongoDB 2013. The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js. Viitattu 26.2.2018, https://www.mongodb.com/blog/post/the-mean-stack-mongodb-expressjs-angularjs-and.

Mongoose 2011. Viitattu 14.4.2018, http://mongoosejs.com/.

Mozilla 2018. Javascript. Viitattu 1.3.2018, https://developer.mozilla.org/en-US/docs/Web/JavaScript.

NoSQL 2018. NoSQL databases explained. Viitattu 27.2.2018, https://www.mongodb.com/nosql-explained.

Node.js. Viitattu 28.2.2018, https://nodejs.org/en/.

npm. Viitattu 9.4.2018, https://npmjs.com.

Pan, Jiantao 1999. Software Testing. Carnegie Mellon University. Viitattu 28.2.2018, http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/.

Passport. Viitattu 14.4.2018, http://www.passportjs.org/.

Regular-Expressions.info 2016. Etusivu. Viitattu 14.4.2018, https://www.regular-expressions.info/.

Request 2018. Github. Viitattu 8.5.2018, https://github.com/request/request.

Rouse, Margaret 2008. object-relational mapping (ORM). TechTarget. Viitattu 9.4.2018, https://searchwindevelopment.techtarget.com/definition/object-relational-mapping.

Saleh, H. 2013. JavaScript Unit Testing. Packt Publishing.

Sinon.JS 2018. Viitattu 5.5.2018, http://sinonjs.org/.

Spies – Sinon.JS 2018. Viitattu 8.5.2018, http://sinonjs.org/releases/v1.17.7/spies/.

Software Testing Fundamentals 2018. Unit Testing. Viitattu 16.4.2018, http://softwaretestingfundamentals.com/unit-testing/.

Stubs – Sinon.JS 2018. Viitattu 8.5.2018, http://sinonjs.org/releases/v1.17.7/stubs/.

Tutorialspoint 2018. Design Patterns – MVC Pattern. Viitattu 14.4.2018, https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm.

```
'use strict';

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const userSchema = new Schema({
  email: {
    type: String,
    required: true,
    unique: true
  },
  forename: String,
  surname: String,
  hash: String,
  salt: String
});

module.exports = mongoose.model('User', userSchema);
```

```
{
    "all": {
        "success": {
            "res": {
                "statusCode": 200,
                "headers": {
                    "content-type": "application/json"
                }
            },
            "body": {
                "status": "success",
                "data": [
                    {
                        "id": 1,
                        "email": "t2koil00@students.oamk.fi",
                        "forename": "Ilkka",
                        "surname": "Koiste"
                    },
                    {
                        "id": 2,
                        "email": "k7koil00@students.oamk.fi",
                        "forename": "Ilkka",
                        "surname": "Koiste"
                    }
                ]
            }
        }
    },
    "single": {
        "success": {
            "res": {
                "statusCode": 200,
```

```
            "headers": {
                "content-type": "application/json"
            }
        },
        "body": {
            "status": "success",
            "data": [
                {
                    "id": 1,
                    "email": "t2koil00@students.oamk.fi",
                    "forename": "Ilkka",
                    "surname": "Koiste"
                }
            ]
        }
    },
    "failure": {
        "res": {
            "statusCode": 404,
            "headers": {
                "content-type": "application/json"
            }
        },
        "body": {
            "status": "error",
            "message": "That user does not exist."
        }
    }
},
"add": {
    "success": {
        "res": {
            "statusCode": 201,
            "headers": {
```

```
            "content-type": "application/json"
          }
        },
        "body": {
          "status": "success",
          "data": [
            {
              "id": 3,
              "email": "ilkka.koiste@mekiwi.org",
              "forename": "Ilkka",
              "surname": "Koiste"
            }
          ]
        }
      },
      "failure": {
        "res": {
          "statusCode": 400,
          "headers": {
            "content-type": "application/json"
          }
        },
        "body": {
          "status": "error",
          "message": "Something went wrong."
        }
      }
    },
    "update": {
      "success": {
        "res": {
          "statusCode": 200,
          "headers": {
            "content-type": "application/json"
```

```
            }
        },
        "body": {
            "status": "success",
            "data": [
                {
                    "id": 1,
                    "email": "ilkka.koiste@mekiwi.org",
                    "forename": "Ilkka",
                    "surname": "Koiste"
                }
            ]
        }
    },
    "failure": {
        "res": {
            "statusCode": 404,
            "headers": {
                "content-type": "application/json"
            }
        },
        "body": {
            "status": "error",
            "message": "That user does not exist."
        }
    }
},
"delete": {
    "success": {
        "res": {
            "statusCode": 200,
            "headers": {
              "content-type": "application/json"
            }
```

```
        },
        "body": {
            "status": "success",
            "data": [
                {
                    "id": 1,
                    "email": "t2koil00@students.oamk.fi",
                    "forename": "Ilkka",
                    "surname": "Koiste"
                }
            ]
        }
    },
    "failure": {
        "res": {
            "statusCode": 404,
            "headers": {
                "content-type": "application/json"
            }
        },
        "body": {
            "status": "error",
            "message": "That user does not exist."
        }
    }
  }
}
```